Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ACCEPTED AT JOURNAL OF SOFTWARE EVOLUTION AND PROCESS

Semantic Similarity Loss for Neural Source Code Summarization

Chia-Yi Su | **Collin McMillan**

[1]Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, Indiana, USA

Correspondence
Corresponding author Chia-Yi Su, Holy Cross Dr, Notre Dame, 46556, Indiana, USA.
Email: csu3@nd.edu

Present address
Holy Cross Dr, Notre Dame, 46556, Indiana, USA.

Abstract

This paper presents a procedure for and evaluation of using a semantic similarity metric as a loss function for neural source code summarization. Code summarization is the task of writing natural language descriptions of source code. Neural code summarization refers to automated techniques for generating these descriptions using neural networks. Almost all current approaches involve neural networks as either standalone models or as part of a pretrained large language models e.g., GPT, Codex, LLaMA. Yet almost all also use a categorical cross-entropy (CCE) loss function for network optimization. Two problems with CCE are that 1) it computes loss over each word prediction one-at-a-time, rather than evaluating a whole sentence, and 2) it requires a perfect prediction, leaving no room for partial credit for synonyms. In this paper, we extend our previous work on semantic similarity metrics to show a procedure for using semantic similarity as a loss function to alleviate this problem, and we evaluate this procedure in several settings in both metrics-driven and human studies. In essence, we propose to use a semantic similarity metric to calculate loss over the whole output sentence prediction per training batch, rather than just loss for each word. We also propose to combine our loss with CCE for each word, which streamlines the training process compared to baselines. We evaluate our approach over several baselines and report improvement in the vast majority of conditions.

KEYWORDS
source code summarization, neural models, optimization, loss functions, human ratings and feedback

1 | INTRODUCTION

Source code "summaries" form the basis for programmer documentation of software. A summary is a natural language description of the behavior of a section of source code. Even a short summary such as "reads list of music files and plays them over speakers" gives a programmer an idea of the purpose of a section of code without ever needing to read the code itself. The process of writing summaries is called source code summarization[1]. The expense of code summarization leads programmers to avoid it and drives strong research interest into automating the process. The ability to write natural language descriptions of source code on demand has long been a dream of software engineering researchers[2], and recent progress in neural source code summarization has drawn this dream within reach.

Neural code summarization refers to code summarization techniques based on neural networks. Almost all current related research uses neural models in one way or another (see Section 2). The workhorse of most approaches is the encoder-decoder architecture in which the source code is "encoded" and a summary is "decoded" as a prediction of the model. Recent advances in academia have focused on the encoding process, such as representing the code as an abstract syntax tree (AST)[3], via a graph neural network (GNN)[4], or customized attention networks[5,6]. In contrast, advances in industry have tended to come from scale and specialized fine-tuning. Large language models (LLMs) including GPT-4, Codex, and LLaMA have demonstrated abilities in explaining and summarizing source code.[7,8].

Yet the training and fine-tuning procedures for practically all of these approaches are based on categorical cross-entropy loss (CCE), which does not always reflect how a human would grade model performance. This loss function is calculated at the end of each prediction and is used to update the model during training. The family of CCE calculate loss for each word as it is predicted, one word at a time. i.e., if the model has predicted "reads list of" so far and next predicts the word "sound" instead of the reference "music", CCE will compute the loss solely on that word being incorrect. The model will be penalized solely

because of that one wrong word, rather than a holistic view of "sound" being similar to "music" in the context of the sentence predicted so far. Wieting et al.[9] refer to this flaw as the loss function lacking the ability to offer "partial credit." The result is that the loss function does not reflect how humans view the output. Unlike CCE and similar functions, humans tolerate mistakes of words or grammar as long as the meaning of the sentence is unchanged.

In this paper, we evaluate the use of semantic similarity as a loss function for code summarization. Specifically, we focus on the semantic similarity metrics between reference summary and the predicted summary. First, we extend our previous work on semantic similarity metrics[10] to show how to use semantic similarity as a loss function during training. In our previous work, we introduced a metric called USE for evaluating code summaries during model testing (after training only). Including USE as part of the training process is much more complicated, though potentially much more rewarding as work in other domains shows[9]. Yet, we do not merely duplicate previous training procedures for a software engineering problem: an additional contribution is that, unlike semantic similarity loss functions in other domains, our loss function is a drop-in replacement for CCE. Existing baselines require an additional reinforcement learning step which adds complexity and reduces uptake by the community. Throughout this paper, we refer to our USE-based loss function as **`use-seq`**.

We present our evaluation in two experiments. First, we train different neural code summarization techniques under standard conditions (i.e., using CCE) and with our USE-based loss function and compare them using automated metrics such as BLEU, METEOR, and USE. Second, we conduct a study with human experts who rate predicted outputs from our approach and from the model trained with the baseline CCE.

Note we conduct our experiments using several academic source code summarization models, but also an approach of our own design using fine-tuning of an industrial LLM. The advantage to using academic models is that we can control key experimental variables such as the contents of the training set. However, a disadvantage is that their small size means they tend to underperform compared to large, industrial models. So, to evaluate if our idea is still relevant at large scale, we fine-tune the LLaMa 7B model by Touvron et al.[11] using the LoRA procedure[12,13,14]. This fine-tuned LLM is an additional novel contribution of this paper. We compare and contrast using CCE, two other baselines, and use-seq loss. We find that our approach benefits both the compact academic models and large industrial ones.

Novelty Statement The key novel contribution of this paper lies in: 1) our procedure for using a semantic similarity metric as a loss function instead of only an evaluation metric, and 2) our evaluation of the semantic similarity loss for the software engineering task of code summarization. This paper is built on our own previous work of semantic similarity[10], and is inspired by semantic similarity as loss in other domains[9]. However, this paper makes a novel contribution with a new procedure and evaluation for code summarization. Our procedure is *not* a duplicate of semantic similarity loss from other domains. As Section 6 will show, we make adjustments specific to the software engineering domain that are not necessarily optimal in a general purpose natural language task, yet are preferred by programmers in experiments.

2 | BACKGROUND AND RELATED WORK

This section discusses key related work and background.

2.1 | Source Code Summarization

Source code summarization is the task of generating short, natural language descriptions of source code. The term "code summarization" was coined by Haiduc et al.[1] in 2010 and the topic has been an active research area since. Between 2010 and 2017, most approaches were IR and template-based[37]. From 2017 to present, neural models have proliferated. Table 1 shows selected papers in the last five years. The progression has been to larger models that explicitly include the code context. Different families have formed, namely AST-based such as ast-attendgru[3], codegnngru[4], transformer approaches[19], and setransformer[26].

Recently, LLM-based dialogue systems such as ChatGPT or tools such as Github Copilot have shown potential in describing code behavior[38,39]. While these tools are not directly comparable because the underlying training data and code analysis procedures are proprietary, they are represented by LLMs fine-tuned on code summarization tasks, as we will show in the following sections. Note that our goal in this paper is to present a loss function for improving code summarization across the board, rather than claiming one single "best" neural model. Thus, our experiments cover different approaches that model code summarization as fine tuning of LLMs (see Section 4). The idea of fine tuning a large language model to summarize code has been presented before[40,41,30,28,29,20,24] – this paper builds on the idea with an improved procedure that trains the model with semantic similarity.

2.2 | Semantic Similarity Metrics

Semantic similarity metrics are the automatic metrics to evaluate the source code summary. The vast majority of the automatic metrics nowadays focus on evaluating the quality between reference summary and the predicted summary such as BLEU[42], METEOR[43],CrystalBLEU[44], and USE[10], which this paper relies on. Another category is to compute the similarity between source code and the summary, which benefits the case where we do not have the good reference summary. For example, Mastropaolo et al.[45] proposed SIDE that gives the summary similar to the source code 1.0 and the summary dissimilar to the source code -1.0 based on contrastive learning. This paper focuses on the metrics that compare the reference summary and the predicted summary and uses the USE as an example. We leave the other metrics as our future work.

2.3 | Loss Functions in Neural Networks

Neural models of code summarization (like models for most NLP applications), predict code summaries one word at a time. During training, the network is run several times with several inputs in a "batch." The concept of batch is to divide the entire dataset into several small groups because we cannot fit the entire dataset into the GPU. In most code summarization approaches, all the words for a summary for a code sample are sent in the same batch (other code samples/summaries may be in that same batch, too). Then the loss is computed over the entire batch and used to update the network. The loss function used in, to our knowledge, all published code summarization techniques is Categorical Cross-Entropy (CCE) loss. CCE uses two values to compute loss over each word in the batch: 1) the value in the reference for that word, and 2) the network's output at each word prediction. The value in the reference is usually a one-hot vector (denoted y) that is the length of the vocabulary size (denoted C). The network's output is also a vocab-size-length vector (denoted $\hat{y}$), adjusted with softmax to represent the predicted probability for the word at each element location, on a 0-1 scale. The loss function formula is then:

$$\mathrm{CCE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

Since $\boldsymbol{y}$ is usually a one-hot vector, for any given word the formula usually simplifies to $\log \hat{y}_r$, where r is the position of the reference word. We show hypothetical CCE values in Table 2 while illustrating our approach.

	T	A	C	F	
Alon et al. (2019)[15,16]	x	x			
LeClair et al. (2019)[3]	x	x			
Nie et al. (2019)[17]	x				
Haldar et al. (2020)[18]	x	x			
Ahmad et al. (2020)[19]	x				
Haque et al. (2020)[5]	x		x		
LeClair et al. (2020)[4]	x	x			
Feng et al. (2020)[20]	x			x	
Bansal et al. (2021)[21]	x		x		
Zügner et al. (2021)[22]	x				
Liu et al. (2021)[23]	x	x			
Mastropaolo et al. (2021)[24]	x				x
Kuang et al. (2022)[25]	x	x			
Tang et al. (2022)[6]	x	x			
Li et al. (2022)[26]	x	x			
Khan et al. (2022)[27]	x			x	
Ahmed et al. (2022)[28]	x		x	x	
Gu et al. (2022)[29]	x			x	
Su et al. (2023)[30]	x			x	
Gao et al. (2023)[31]	x	x			
Geng et al. (2023)[32]	x		x		
Zhang et al. (2023)[33]	x	x			
Gao et al. (2023)[34]	x	x			
Wang et al. (2023)[35]	x	x	x		
Geng et al. (2024)[36]	x		x	x	

TABLE 1 Snapshot of the past five years in source code summarization. Column *T* means use of source code as Text. *A* signifies learning from AST. *C* implies learning chiefly from code context. *F* means primary a fine-tuning approach for an LLM.

2.4 | Semantic Similarity Loss

Alternatives to CCE have been proposed that compute loss based on semantic similarity. Two approaches stand out as particularly relevant to code summarization: 1) using the n-gram sequence similarity metric BLEU to compute loss[46,47], and 2) SimiLE, which uses cosine similarity of an embedding vector model to compute loss[9]. SimiLE represents a family of techniques based on embedding vector similarity[48,49,50]and includes improvements such as a length penalty.

We expand on more details of these approaches in Section 4.2, but the key advantage over CCE is that they calculate loss over an entire output sequence prediction instead of each individual word. However, a key disadvantage is that they require an additional reinforcement learning phase after normal CCE training that adds complexity and experimental variables. In contrast, our approach is a drop-in replacement for CCE, and our approach is designed and evaluated for the domain-specific SE task of code summarization. As we will show in Section 6, adjustments useful in general natural language such as a length penalty are not necessarily ideal for code summarization, where programmers tend to value accuracy over conciseness.

3 | APPROACH

This section discusses our approach to the semantic similarity loss function we propose called use-seq. The use-seq calculation is broadly divided into six steps. We show concrete outputs for each word at each step in Table 2.

TABLE 2 Example of loss calculation for each word at each step. The batch loss is the average loss from each step. CCE values are hypothetical for illustration.

predicted word	step 3	step 4	step 5	(cce)	step 6
records	0.8665	0.8665	2.954	0.0400	0.1182
a	0.8665	0.8665	2.954	0.0500	0.1477
sound	0.8665	m	m	0.8000	0.8000
file	0.8665	0.8665	2.954	0.0600	0.1772
				batch loss:	0.3108
				USE score:	0.8665

1) Convert predicted sequence into natural language. During training, each training batch will consist of at least one example subroutine and summary. Then the model will predict each position of the output one-at-a-time using the "teacher forcing" procedure. For example, consider a subroutine with the reference summary "records a music file", but where the model makes an error by predicting the word "sound" instead of "music". We demonstrate the prediction process for this sample in Table 3.

TABLE 3 Demonstration of prediction process

	training input	reference word	predicted word
1	<s>	records	records
2	<s> records	a	a
3	<s> records a	music	sound
4	<s> records a music	file	file
5	<s> records a music file	</s>	</s>

The model will receive the reference training input, so that the last word "file" is predicted using the correct previous word "music." However, the total predicted sequence still contains the error. In CCE, the loss is calculated for each predicted word to the reference word. In our `use-seq` approach, we convert the predicted words back into a sequence, in this case "records a sound file." The special tokens <s> and </s> are the start and the end tokens. The adoption of those tokens in the example is to show the entire process of the model prediction.

2) Compute semantic similarity. The next step is to take each predicted and reference sequence, obtain the USE vector for each sequence, then compute the cosine distance between the two vectors. The USE vector is a 512-length vector from the universal sentence encoder model[51]. Technically, we could use any word sequence encoder to produce these vectors, but a recent paper finds that USE produces results most in line with human preferences for code summarization[10].

3) Broadcast semantic similarity to each word. The semantic similarity calculation in `use-seq` applies to the entire sequence. However, loss is usually calculated for each individual prediction, which in our case is each word. We assign the loss for each word to be equal to the loss for each sequence. For example, the cosine distance between the reference and predicted sequences above is 0.8665, so each word "records", "a", "sound", "file" will receive a loss of 0.8665.

4) Mask semantic similarities to avoid inappropriate penalties. It is possible that USE will return a score that indicates a strongly dissimilar predicted sequence, even when some individual word predictions may be correct. It is also possible that USE will indicate a similar sequence, even when individual word predictions may be incorrect. For example, "disconnect db" and "initialize database connections" are two opposite sentences, but we had 0.5613 as the USE for these two sentences. Although this score is lower compared with "connect db" and "initialize database connections," which we had 0.6963 as the USE score, 0.5613 is still very high for the opposite sentence. In these situations, a naive application of sequence similarity to each word could have the effect of rewarding the model for incorrect word predictions or penalizing the model for correct word predictions. To avoid these problems, we create a mask in which, for each word prediction, if the prediction is correct, only use the sequence similarity broadcast to that word if the cosine similarity is positive. Likewise, if the prediction is incorrect, only use the sequence similarity broadcast if the cosine similarity is negative. The effect is to provide an extra reward to correct predictions when the sequence similarity is high, while also giving a penalty to incorrect predictions when the sequence similarity is low. In Table 2, "m" denotes a mask for the word "sound" because "sound" is incorrect even though the overall sentence similarity is high as indicated in Table 2 that the USE score is 0.8665.

5) Adjust semantic similarities using exponentiated reward. In preliminary experiments, we noticed that semantic similarity values tend to be distributed such that small differences from the mean seem to have less overall meaning than large ones. In other words, semantic similarity is most useful in reporting that a sequence is very similar or dissimilar, while values around the mean are harder to interpret. Our observation corroborates findings in using human ratings of similarity, such as by Korbak et al.[52], and our remedy is similar: we adjust each semantic similarity score using an exponential function adjusted by parameter β. The formula is $exp(R(x_i)/\beta)$, where x_i is the word prediction for position i in the sequence, and R is the reward function, which in our case is the USE similarity score for the sequence where x_i originates. The effect of this function is to push values beyond a certain threshold to have much more effect on the loss. The value β is a parameter which allows us to scale the similarity scores. For example, in Table 2, the scaled reward value becomes $exp(0.8665/1) = 0.8665$ when *beta* is 1. We use $\beta = 1.0$ as a default value for LLMs because it is recommended by Korbak et al.[52]. However, we explore the different values in Section 4.7. We use $\beta = 0.8$ as a default value for purpose-built models because our ablation study finds that $\beta = 0.8$ shows the improvement over all datasets and across all metrics and datasets.

6) Combine the semantic similarities with CCE for each predicted word. A problem we noticed in preliminary experiments was that using semantic similarity scores as a loss in from-scratch training leads to very unstable and poor results – an observation also found in using semantic similarity for other domains[50]. The solution in related work is to train the model to convergence using CCE, then add a fine-tuning step with semantic similarity loss. The problem with adding a fine-tuning step is added complexity of the training procedure and creation of new experimental variables (how far to train after convergence with CCE, what parameters/methods of fine-tuning, etc.). Our solution is instead to use semantic similarity to adjust CCE for each word (alluded to in Step 4). We multiply the CCE for each word to the semantic similarity score for that word. We formalize entire procesure in Equation 1.

$$R(x_i) = \begin{cases} USE & \text{if } x_i \text{ is correct} \\ 0 & \text{if } x_i \text{ if incorrect} \end{cases}$$

$$\text{loss}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{C} y_i \log \hat{y}_i * exp(R(x_i)/\beta) \tag{1}$$

Since $\boldsymbol{y}$ is usually a one-hot vector, for any given word the formula usually simplifies to $\log \hat{y}_r$, where i is the position of the word; $R(x_i)$ is the reward function; β is the hyperparameter. We show hypothetical CCE values in Table 2 while illustrating our approach.

While the description above is sufficient to reproduce our approach, given its complexity we encourage other researchers to read our implementation in the function `custom_use_seq()` at line 191 of file `custom/qs_loss.py` in our reproducibility package (Section 9).

4 | EXPERIMENT

This section describes our controlled experiment involving code summarization models and automated metrics. Note that this experiment is distinct from our human study in Section 6.

4.1 | Research Questions

Our research objective is to test whether semantic similarity loss improves model training results for the task of source code summarization. We formalize our experiments based on Wohlin et al.[53]. We define the independent variable as the loss function in the model and the dependent variable as the results in terms of the automatic metrics i.e. BLEU, METEOR, and USE. The hypothesis is that `use-seq` should perform better than any other loss functions in any circumstance e.g. `use-seq` should have better performance than CCE in both purpose-built models and LLMs. Towards that end, we ask the following Research Questions (RQs):

RQ1 What are the differences in performance among `use-seq` and the baselines for purpose-built source code summarization models?

RQ2 What are the differences in performance between `use-seq` and the primary baseline for select Large Language Models (LLMs) fine-tuned for source code summarization?

The rationale behind RQ1 is that our approach should benefit several neural network-based models of code summarization among the many that have been published in recent years. These models tend to be relatively small (on the order of 30m-50m parameters), but are purpose-built under highly-controlled experimental settings that are available to the public. While the performance may or may not be as high as industrial solutions such as ChatGPT or Copilot, the advantage is that we can be more certain of the effect of different experimental variables when we have access to all model details and training data.

The rationale behind RQ2 is that our approach should, in theory, benefit any model type based on neural networks, including very large ones. There are several approaches to use the pretrained Large Language Models (LLMs) for downstream tasks. We can either use prompt techniques or finetuning the models for the tasks. The advantage of those approaches is that it allows for the use of much bigger models (100m up to many billions of parameters) that tend to have better results, at least in terms of automated metrics. This paper mainly focuses on the finetuning method because our main goal is to show that the model trained with `use-seq` improves the training results.

However, the disadvantage is that the pretraining process is so expensive that it is constrained to large industrial organizations[54]. Many details such as the training data sets are closed source – a major hazard for research because the training set may contain some or all of the test set[55]. So, we ask RQ2 to test our approach in a fine-tuning setting to be current with the latest techniques, but we retain RQ1 as a balance in a setting where we have maximum control over experimental variables.

4.2 | Baselines

There are three baselines for our work in our experiment: CCE, BLEU, and SimiLe. Categorical Cross-entropy (CCE) loss, as we established in Section 2, is by far the main means of computing loss for text generation tasks, including code summarization. It represents the state-of-the-practice. BLEU is usually used as a metric for evaluation, but has been proposed as a loss function for text generation[46,47]. Using BLEU in this fashion represents an n-gram text similarity metric. Finally, SimiLe is a semantic text similarity metric repurposed as a loss function[9]. The technique is the most similar approach in the literature to this paper. It combines a vector-based text similarity technique (SIM by Wieting and Gimpel[56]) with a length error penalty component. Note that in this paper, we replaced SIM with USE as the text similarity foundation for SimiLe. Compared with SIM, USE is newer, trained on a large corpus of text, and has a highly-supported implementation for reproducibility. In addition, using USE reduces experimental variables as we can determine that the difference between SimiLe and other approaches is more likely to be due to the loss function formula instead of differences in the text similarity calculation.

Note that BLEU and SimiLe rely on a reinforcement learning-like training procedure that is more complex than the one needed for our approach. As Wu et al.[57] and Yasui et al.[50] point out, using a text similarity metric out-of-the-box as a loss function tends to lead to very unstable training results. Therefore, the recommended procedure is to train using CCE to allow the

model to converge, then fine tune using the semantic similarity loss. In our experiments, we report BLEU and SimiLe results after one epoch of fine tuning after the CCE loss convergence. Since the number of samples in our datasets is small (as opposed to internet-scale LLM training), we use the same learning rate and other hyperparameters during the fine-tuning epoch.

In contrast, an advantage of our approach is that we do not require the additional fine tune epoch and related procedural complexity. Our approach is a drop-in replacement for CCE and we train models using it in the same way.

4.3 | Datasets

We use three datasets. The first dataset, named `funcom-java`, consists of about 2 million Java methods. The dataset was originally introduced by LeClair and McMillan[58], who advocate a split-by-project configuration to avoid data redundancy. We used the updated version of the dataset introduced by Bansal et al.[21], who applied additional filtering techniques to remove code clones as suggested by Allamanis[59].

We also generated a subset of the above Java dataset, which we call `funcom-java-long` , to focus on the methods that have the higher number of code tokens and implement the key data preprocessing filters suggested by Shi et al.[60]. It contains the subroutines that have top 10% highest number of code tokens. The reason that we focus on these functions is based on the observation made by Haque et al. [61] that many Java functions are trivially short such as getters and setters. The focus of the methods that have a higher number of code tokens can show that our approach is able to tackle more challenging and realistic methods because these methods are harder to understand and have less training data.

Lastly, we compiled a dataset of Python functions from 40,000 Python projects we downloaded from GitHub, named `funcom-python`. We employed the same preprocessing and splitting methods as recommended by LeClair and McMillan[58] and Bansal et al.[21] to create a dataset of 270k functions.

4.4 | Metrics

We use the metrics METEOR, USE, and BLEU. BLEU by Papineni et al.[42] is an n-gram based text similarity metric used for over twenty years in several areas of research including code summarization. METEOR by Banerjee and Lavie[43] updates the idea of BLEU to include the semantic similarity of each word. METEOR is preferred to BLEU for code summarization in light of recent evidence finding that METEOR is more correlated to human judgment[10,62]. USE is the semantic similarity metric proposed by Haque et al.[10] that is the basis for our semantic similarity loss functions, described in Section 2.

We calculated statistical significance using a paired t-test for METEOR and USE between `use-seq` and the baselines for each code summarization technique, using the procedure suggested by Roy et al.[62] for code summarization. However, we do not calculate statistical tests for BLEU because BLEU is a corpus-level metric and not considered reliable when calculated at sentence-level[42]. Space limitations prevent us from printing full results in this document, so we report when the test was not significant at the $p > 0.05$ level with an asterisk in Table 5.

4.5 | Code Summarization Models

We answer our RQs in the context of four purpose-built code summarization models. These models represent different families of models that we identified in Section 2. In our view, these approaches serve as "mouse models" which may have some distance from in-practice use, but have the major advantage that we can tightly regulate experimental variables including complete training in the laboratory setting and reproducibility at reasonable cost. The four models are:

ast-attendgru An approach that encodes the target function's Abstract Syntax Tree (AST) via a flattened representation and a recurrent neural network (a GRU)[3].

codegnngru An approach by LeClair et al. [4] that is similar to `ast-attendgru` except that it uses a graph neural network (GNN) to encode the AST.

transformer Essentially this approach uses a vanilla Transformer encoder-decoder design, proposed for use on code summarization by Ahmad et al.[19].

setransformer A hybrid Transformer-CNN model proposed recently by Li et al.[26] that encodes the AST and textual information from the code.

We train all four models from scratch on the training set from each dataset. Our main interest is to test our loss function rather than other variables, so we follow the training procedure established by several recent papers[21,5,4]: train for ten epochs, select the epoch for which the validation accuracy was the highest, then report metric scores over the testing set for that epoch. Key hyperparameters are shown in Table 4:

TABLE 4 Training hyperparameters and settings

		Java	Python
t	tokens in target subroutine	50	50
w	words in summary	13	13
v	source code vocabulary size	75k	100k
z	summary vocabulary size	10908	11000
e	embedding dimensions	100	100
b	batch size	50	50
r	learning rate	1e-4	1e-4
o	optimizer	Adam	Adam

We used t and w reported by Haque et al.[5] and Bansal et al.[21]. The values for v and z are suggestions from a study of code summarization datasets[58]. We decided e, b, r, and o during pilot studies and constrained by hardware limitations – our goal is for our experiments to be reproducible with moderately-priced professional hardware.

4.6 | Fine-tuned Large Language Models

We also answer our RQs in the context of fine-tuned large language models (LLMs). State-of-the-art performance in many text generation tasks is often produced by fine-tuning so-called foundation models. A foundation model is an LLM that is pre-trained on internet-scale text datasets. Then the foundation model is fine-tuned by further training on a relatively small dataset of domain-specific examples. There are hundreds of possible fine-tuning configurations, and a comprehensive study of fine-tuning for source code summarization is not available yet in the literature and is beyond the scope of this single paper. However, our goal is to determine the usefulness of our semantic similarity loss function in a wide range of models, so we chose two approaches consistent with related work for other text generation tasks. Given the high cost of fine-tuning LLMs and the immense number of experimental variables given the rapidly changing research frontier, we limit ourselves to comparing `use-seq` to CCE for the `funcom-java-long` dataset.

One LLM we use is the LLaMA 7B parameter model by Touvron et al.[11]. We fine-tune this model with the Alpaca-LoRA procedure using the settings and implementation available from Taori et al.[12]. Technically, we set the `Instruction` text to "please describe the following source code", the `Input` text to the target function's source code, and the `Response` text to the source code summary during fine tuning. Then we fine tune for one epoch using the default parameters (listed in our reproducibility package, Section 9). During inference, we use the same `Instruction` and `Input` text setup, but extract the text after `Response` as the summary.

We also use the GPT2 124m parameter model by Radford et al.[63]. We use a complete fine-tune procedure (as opposed to a weight matrix reduction technique such as LoRA) based on OpenAI's GPT2 124m parameter snapshot. We use hyperparameters recommended by Karpathy[64] in a public GPT2 implementation. We fine tune for 18 epochs, at which point validation accuracy diverges from training accuracy, indicating possible overfitting. We acknowledge that these parameter choices are somewhat arbitrary, but our objective is to test the effect of training with CCE versus `use-seq`, rather than an exhaustive search for all optimal parameters. Our idea was to use reasonable settings from related work, and compare performance differences when changing only a single experimental variable: the loss function.

In addition to 124m parameter GPT2, we use 220m CodeT5+[40] with complete finetuning procedure to show that our `use-seq` loss can be applied to broader language models i.e. the large language model with the encoder-decoder architecture. We finetune for three epochs. We use the hyperparameters set in the finetuning script in the CodeT5+ repository. We use the same hyperparameters for both CCE and `use-seq` because our main goal is to show that the model trained with `use-seq` is better than the model trained with CCE loss.

Note that we intend for these models to be representative of applying the technique to fine-tuning commercial LLMs, to the extent possible in a research setting where we exert control of experimental variables and are able to open source release

experimental artifacts. As Hellendoorn and Sawant[54] point out, large, closed source LLMs such as GPT3.5 or GPT4 produce strong results at the expense of accessibility of the internals to the research community. Smaller models capable of being run in-house also have the major advantage of not requiring proprietary code data to be sent to a third party. The loss of data custody required to use e.g. GPT4 is not tenable for many organizations. Furthermore, emerging evidence is showing that commercial LLMs do not always provide stable results, making them difficult to benchmark in a controlled experiment[38,39]. Therefore, we evaluate our idea in the situation of in-house, open source production.

4.7 | Ablation Study

To show the necessity of the reward mechanism that we introduce in Section 3 and study the different β value, we conduct the ablation study with the GPT2 model. We used the GPT2 model because this model does not require weight matrix reduction technique to finetune. Also, this model has the best performance among the models that do not need weight matrix reduction technique. As an additional verification, we also used the dataset proposed by Su and McMillan[65] to finetune the GPT2 model with the parameters that we introduce in Section 4.6. Specifically, we focus on 170k dataset because it has exactly the same method as in `funcom-java-long`. However, instead of obtaining summary from human programmers, the summary of this dataset is generated by using GPT-3.5.

4.8 | Hardware/Software Details

Our hardware platform is a workstation with an AMD 5900X CPU, 128GB memory, and two Nvidia A5000 GPUs. Our software platform consists of Ubuntu 22.04, Python 3.10, CUDA 11.4, Tensorflow 2.9.1, and Pytorch 2.0.0.

5 | EXPERIMENTAL RESULTS

This section discusses our experimental results and answers to research questions RQ1 and RQ2.

5.1 | RQ1: Purpose-built Models

Table 5 summarizes our experimental results for RQ1. In short, we find that `use-seq` outperforms the baselines in most conditions over the three datasets and four purpose-built code summarization models. We observe the strongest overall performance in the `transformer` model, with 34.16 METEOR and 52.23 USE scores in the `funcom-java-long` dataset. These are about a 2% and 3% improvement over SimiLE and CCE for these metrics in `funcom-java-long`. In `funcom-java` and `funcom-python`, the improvements are mostly in the 1-2% range.

It is important to note that these improvements come at practically no cost, since `use-seq` is a drop-in replacement for CCE. These improvements cover a broad spectrum, as we observe them over datasets with two different programming languages and over several model architectures. So, even a modest increase in metrics scores can have a big impact on the state-of-the-art because the increases benefit many approaches at almost no cost.

We observe higher rates of improvement using BLEU score, with increases in the 2-7% range for `transformer` over the three datasets. The highest performing model in terms of BLEU is `ast-attendgru` with `use-seq` over `funcom-python`, which at 20.12 BLEU is 12% higher than the same model using SimiLE and 18% higher than CCE. We attribute this difference between BLEU and METEOR/USE to BLEU's use of exact-match n-grams versus METEOR/USE's word similarity approach. Our approach includes a mask that scales the reward based on sentence similarity, but retains a high penalty for incorrect words even in "good" sequence predictions (Section 3, step 4). In contrast, the BLEU and SimiLE functions reward the model when the overall sequence is "good", even when individual word predictions are incorrect. The result is that our approach gets more individual word predictions correct, and this result manifests itself as bigger gains for BLEU than METEOR or USE.

Two exceptions to the general rule of improvement are: 1) METEOR/USE scores for `setransformer` over `funcom-python`, and 2) a handful of results that lack statistical significance, especially for `codegnngru` over `funcom-python`. One explanation is that `setransformer` and `codegnngru` rely more than other models on the code's

AST (even `ast-attendgru`, which has separate encoders), which is often less informative in Python (e.g., due to dynamic typing), which was also observed by Tang et al.[6].

TABLE 5 Results of automatic metrics for RQ1. M=METEOR, U=USE, B=BLEU. W is the number of times `use-seq` was the highest over all metrics and datasets. An asterisk indicates METEOR or USE results that are not statistically different from `use-seq` according to a paired t-test at the $p < 0.05$ level.

		funcom-java-long			funcom-java			funcom-python			
model	loss	M	U	B	M	U	B	M	U	B	W
ast-attendgru	cce	33.21	50.12	18.94	35.30	52.89	18.33	26.48	43.27	16.96	
	bleu	33.43	50.02	18.92	35.56	53.03	18.68	26.41	42.37	17.64	
	simile	33.34	49.87	18.87	35.68	53.18	18.77	26.54	42.66	17.89	
	use-seq	**33.74**	**50.52**	**19.38**	**35.96**	**53.74**	**19.07**	**28.42**	**44.00**	**20.12**	9/9
codegnngru	cce	32.98	49.85	18.75	35.82	53.26	18.77	25.32	41.86	16.80	
	bleu	32.53	49.46	18.66	35.64	53.21	18.61	25.74*	42.14*	**16.93**	
	simile	32.47	49.52	18.66	36.03*	53.43	18.66	25.56*	42.13*	16.81	
	use-seq	**33.97**	**50.92**	**19.51**	**36.17**	**53.84**	**19.25**	**25.89**	**42.17**	16.51	8/9
transformer	cce	33.57	51.60	19.07	35.86	53.89	18.39	26.97	44.02	15.70	
	bleu	33.52	51.60	18.92	35.92	53.84	18.60	26.89	43.62	16.37	
	simile	33.40	51.31	18.90	35.98	53.79	18.63	26.87	43.76	16.35	
	use-seq	**34.16**	**52.23**	**19.63**	**36.17**	**54.45**	**18.97**	**27.38**	**44.31**	**17.56**	9/9
setransformer	cce	32.60	49.56	18.38	35.64	53.09	18.26	27.92	43.48	18.10	
	bleu	32.21	48.80	18.49	35.99	53.36	18.69	28.59*	43.94	18.76	
	simile	32.22	48.66	18.48	36.03*	53.43	18.66	**28.61***	**44.05**	18.91	
	use-seq	**33.51**	**50.79**	**19.04**	**36.35**	**54.14**	**19.12**	28.57	44.02	**19.46**	7/9

5.2 | RQ2: Large Language Models

We find that `use-seq` improves fine-tuning results by a statistically-significant margin for all of the LLM models i.e. GPT2, LLaMA, and CodeT5+ models. Table 6 shows these results for RQ2. For GPT2, we observe a 8% improvement in METEOR, a 4% improvement in USE, and a 8% improvement in BLEU. For LLaMA, we observed improvements of 3% and 5% for METEOR and BLEU. USE was a statistical tie with only a 0.2% increase that was not found to be significant. Although CodeT5+ has the least improvement among these models, we still observe at least 1% improvement with a statistically-significant margin in both METEOR and USE. These results are in broad agreement with findings from RQ1.

These results point to the usefulness of `use-seq` even at scale. The purpose-built code summarization models that we used are in the range of 30-50m parameters[4,19,26], while the GPT2 model we used is 124m parameters, the CodeT5+ is 220m, and the LLaMA model we used is 7B parameters. The purpose-built code summarization models have no pretraining data, and the GPT2, CodeT5+, and LLaMA models each have different sets of pretraining data composed of differing amounts of text, code, and other types of language artifacts. It is likely that additional hyperparameter tuning and model optimization would yield higher overall scores, though the evidence here is that `use-seq` confers an advantage to a wide variety of models under different conditions. Meanwhile, baselines except CCE are not practical for these larger models as they would require at least one additional training epoch (around 60 hours for `funcom-java-long` for LLaMA on our hardware).

TABLE 6 Results of automatic metrics for RQ2. An asterisk indicates METEOR or USE results that are not statistically different from use-seq according to a paired t-test at the $p < 0.05$ level.

		funcom-java-long		
model	loss	M	U	B
gpt2	cce	32.77	51.37	19.02
	use-seq	**35.51**	**53.50**	**20.58**
llama7b	cce	39.60	59.60*	23.22
	use-seq	**40.87**	**59.73**	**24.40**
codet5+	cce	17.33	46.58	**0.23**
	use-seq	**17.53**	**47.20**	0.00

5.3 | Results of Ablation Study

Table 7 summarizes the results of the ablation study for reward mechanism. Overall, the model with the reward mechanism that we introduce shows the strongest improvement across three metrics and two different datasets. Specifically, we find a 5% improvement in METEOR, a 4% improvement in USE, and an 8% improvement in BLEU with the reward mechanism on `funcom-java-long` dataset compared with the model without reward mechanism. For GPT-3.5, we find the 5% improvement in METEOR, 2% improvement in USE, and the 14% improvement in BLEU. Moreover, we find 11% drop in BLEU score when we compare CCE with the model without the reward mechanism on GPT-3.5 summary. These results show the necessity of the reward mechanism to adjust the final reward, so the models do not reward the incorrect prediction and penalize the correct prediction.

Table 8 shows the results for different β values. In short, we find $\beta = 0.8$ has the improvements over three different metrics i.e. BLEU, METEOR, and USE and two different datasets compared with $\beta = 1.0$. Specifically, we observe that $\beta = 0.8$ has the strongest improvement on USE and BLEU, which has a 0.7% improvement on USE and a 2% improvement on BLEU among these different β values on the `funcom-java-long` dataset. Although $\beta = 0.8$ does not have the strongest improvement in terms of GPT-3.5 summary, the improvement between $\beta = 0.8$ and $\beta = 0.6$ is relatively small compared with the improvement between $\beta = 0.8$ and $\beta = 1.0$. The improvement between $\beta = 0.8$ and $\beta = 1.0$ is 1% versus 0.1% improvement between $\beta = 0.8$ and $\beta = 0.6$ in terms of METEOR. These results show that $\beta = 0.8$ is an appropriate choice because it shows improvements over a wide variety of datasets and metrics.

TABLE 7 Result of ablation study on reward mechanism. cce means the model trained with the cce loss. use-seq means the model trained with the use-seq loss with the reward mechanism that we proposed. use-seq-ablate means the usq-seq loss without reward mechanism that we proposed.

dataset	loss	M	U	B
funcom-java-long	cce	32.77	51.37	19.02
	use-seq-ablate	33.76	51.48	19.12
	use-seq	**35.51**	**53.50**	**20.58**
GPT-3.5 summary	cce	33.68	62.42	12.90
	use-seq-ablate	37.43	63.30	11.42
	use-seq	**39.35**	**64.78**	**13.00**

TABLE 8 Result of ablation study on selection of β

dataset	β	M	U	B
funcom-java-long	1.2	35.71	51.99	17.22
	1.0	35.51	53.50	20.58
	0.8	35.68	**53.88**	**20.94**
	0.6	**36.05**	52.29	17.14
GPT-3.5 summary	1.2	39.63	65.00	13.49
	1.0	39.35	64.78	13.00
	0.8	39.83	65.01	13.36
	0.6	**39.90**	**65.31**	**13.70**

5.4 | Examples

We provide four examples in Table 9 to illustrate different scenarios of how the models perform. Example 1 shows a method called `draw()` where `use-seq` resulted in better summaries throughout the experiment. The fine-tuned LLaMA model predicted an addition modifier "control" with `use-seq`, which not only more-closely matched the reference, but provided an extra word that humans reported as being more accurate (see our human study in the next section). Likewise, Example 2 shows how `use-seq` led the LLaMA model to output additional relevant information compared to CCE. In general, the predictions using `use-seq` are longer than with CCE, likely because during training, predictions of the end-of-sequence token will be

penalized especially heavily when they are both wrong and cause the sequence to lack information from the reference. If the predicted sequence is not similar enough to the reference, the USE score will be low (Section 3, step 3), and that low score will amplify the penalty of a mispredicted token such as the end of sequence token (Section 3, step 4).

Examples 3 and 4 show where `use-seq` does not necessarily always help. Example 3 is an oddity because it is an exception to the general rule of more verbose summaries from `use-seq`, as it missed the word "xy" modifying dataset. It also shows how human sometimes prefer the more verbose summaries, even if the metrics scores are lower. Example 4 shows a similar situation for LLaMA, where the more verbose summary gets lower metrics scores, but actually includes more relevant information in the summary. These situations tend to favor `use-seq` because `use-seq` generally leads to more verbose predictions. We will explore the issue of verbosity in summaries more in our human study in the next section.

TABLE 9 The prediction examples. Transformer means of output from the purpose-built model. llama7b refers to the output from finetuned Llama. M=METEOR, U=USE. H=1 means summary was considered more accurate in the human study (dash means summary not shown in human study). The number underneath the method name is the identification number of that method in the `funcom-java-long` dataset, which we provide for reproducibility. Note that for illustration we deliberately chose some examples where `use-seq` does not show improvement.

				M	U	H
1. draw() *7848915*	reference		draw the control fps panel in the control sketch window			
	transformer	cce	draws the graphic	10.75	26.12	-
		use-seq	draws the outline of the current page	**26.35**	**27.36**	-
	llama7b	cce	draw the fps	16.13	37.20	0
		use-seq	draws the control fps	**33.58**	**39.54**	1
2. destroyMainPart() *13278733*	reference		destroys one of the main parts of the given docking graph	M	U	H
	transformer	cce	removes the main part of the graph	55.56	75.73	-
		use-seq	destroys the main part of the graph	**63.44**	**81.56**	-
	llama7b	cce	destroys the main part	30.64	56.77	-
		use-seq	destroy the main part of the docking graph	**72.79**	**82.85**	-
3. createDataset() *35061399*	reference		creates a sample dataset	M	U	H
	transformer	cce	creates a dataset for the chart	**10.64**	10.55	-
		use-seq	creates a dataset	10.00	**36.93**	-
	llama7b	cce	creates an xy dataset	10.64	10.55	1
		use-seq	creates a dataset	**19.94**	**17.68**	0
4. copySubstring() *40467691*	reference		copy string to clipboard	M	U	H
	transformer	cce	copies the substring of the specified	11.90	50.24	-
		use-seq	copies the substring of the code string code	**22.72**	**57.02**	-
	llama7b	cce	copies a substring of the text to the clipboard	**33.33**	**69.99**	-
		use-seq	copies the substring from the given start to the given end to the clipboard	30.00	47.25	-

5.5 | Threats to Validity

We divide Key threats to validity into three different categories i.e. internal, construct, and external threats to validity based on Wohlin et al.[53]. The internal threats to validity include datasets and the pretrainng data used in the LLMs. The datasets are a threat to validity because different test inputs could yield very different results. To help mitigate this risk, we use datasets over two languages (Java and Python), with a special emphasis on subroutines with more code tokens in Java. key risk to the LLMs is that the training data is closed source and we cannot guarantee that the test sets (which are derived from open source projects) are not in the training set. We aim to mitigate this risk by contrasting the LLM part of the experiment from the code summarization models (for which there is no pretraining data) and by using LLMs that are reportedly trained on different datasets. The construct threats to validity is the code summarization models and LLMs we train/fine-tune. The code summarization models, training, and fine-tuning procedures can also affect the results, as e.g., more epochs for training may yield better or worse results. We mitigate this risk by using established experimental procedures in code summarization models and by making diverse choices for the LLMs (e.g., LoRA versus a complete fine-tuning, LLaMA versus GPT2 and CodeT5+) to decrease the risk of the results being meaningful in only a special setting. The external threats to validity is `use-seq` might not be able to be applied to the

most up-to-date LLMs. We mitigate this by conducting the experiments with three different types of LLMs i.e. LLaMA, GPT2, and CodeT5+.

6 | HUMAN STUDY

This section describes our study with human programmers. In short, we recruited human experts to compare the summaries generated by the LLaMA 7B model that we fine-tuned using CCE and `use-seq`.

6.1 | Research Questions

The objective of the human study is to show that `use-seq` not only has better performance in terms of automatic metrics but is preferred by the programmers. The independent variable for this experiment is loss functions. The dependent variable is the results from the human experts. The hypothesis is that the improved summary should reflect on the results of both automatic metrics and the human experts. Therefore, we ask the following research questions:

RQ3 How do human experts rank `cce` and `use-seq` in terms of the quality attributes accuracy, completeness, conciseness, and similarity to a reference?
RQ4 Which of `cce` and `use-seq` do human experts prefer overall, independent of individual quality attributes?

The rationale behind RQ3 is that many years of studies in source code summarization use four quality attributes to compare summaries. These are: 1) Accuracy, which refers to whether the information in the summary is correct, 2) Completeness, which refers to whether the summary contains all the information it should, 3) Conciseness, which refers to whether the summary is overly verbose or contains unnecessary information, and 4) Similarity to a reference, which is a human judgment about whether a summary is similar to the summary for a code in the gold set. These quality attributes have a very long history in evaluating code comments[66,67,68], so we use them as the basis for evaluating our approach.

The rationale behind RQ4 is that human experts may have their own subjective opinion about the quality difference between two summaries, and this opinion may be separate from the quality attributes typically studied in the literature. People are unique, and sometimes may prefer one thing over another for unpredictable reasons. We study RQ4 to capture these unpredictable preferences.

6.2 | Study Design

Our study design centers around a web survey in which participants read the source code of a subroutine and a summary for that subroutine, then answer five questions (see Figure 1). The survey consisted of a tutorial with definitions and examples of the quality attributes, followed by four "pages" per subroutine. On the first page, the people saw the subroutine source code, one summary generated by the model after fine-tuning with CCE, one summary generated by the model after fine-tuning with `use-seq` (order of summaries in UI was random and did not indicate its origin to avoid demand characteristic bias[69]), and the following question:

Q1 Independent of other factors, which summary is more accurate?

The next page showed the same information, but with the following two questions. We asked these questions in a separate page to avoid a bias from showing a positively-worded question alongside a negatively-worded one[70]:

Q2 Which summary is missing more information that is important for understanding the method?
Q3 Which summary contains more unnecessary information?

The next page showed the same information, but with the following question:

Q4 Overall, which summary is better in your opinion?

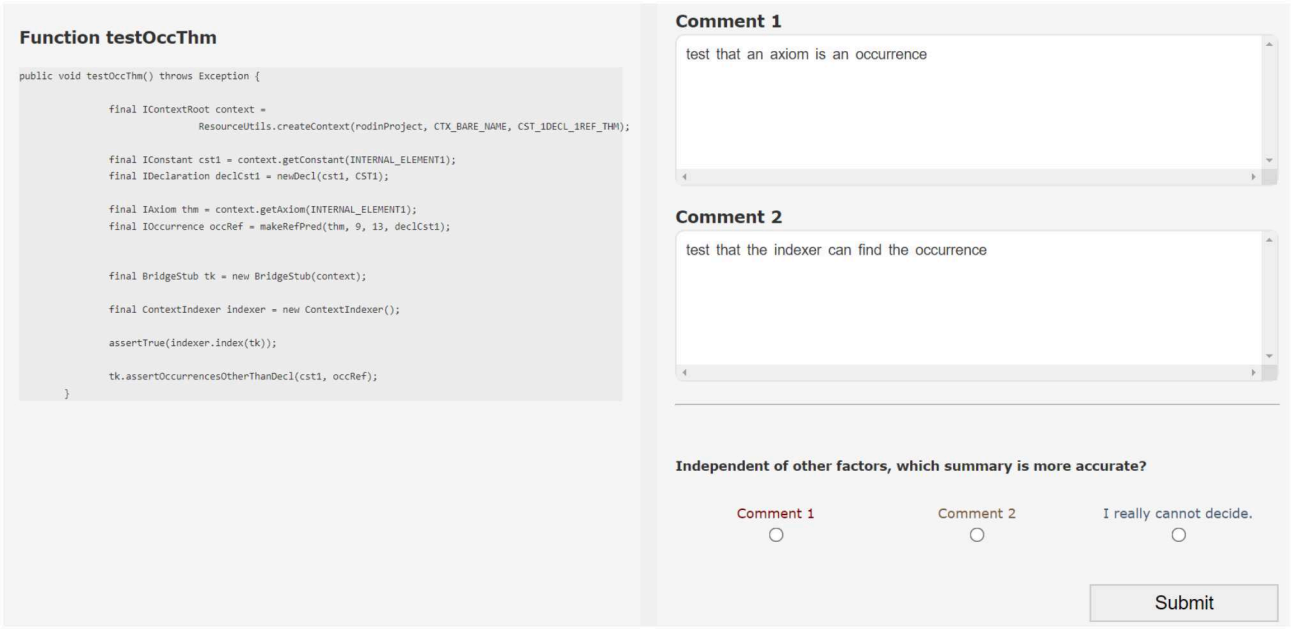


FIGURE 1 Screenshot of page one of the survey interface.

And finally, the last page showed the same information, but with a reference summary provided on the left side of the screen and the following question:

Q5 Which summary is more similar to this third summary on the left?

It is prohibitively expensive to run a human study with all models and datasets from our experiment, so we focus on comparing CCE to `use-seq` in the following conditions:

Model We use the summaries generated by the LLaMA 7B model. We focus on this model because it had the highest performance in our experiment in the previous two sections, and so is likely the model closest what could eventually be used in production.

Subroutines We used 56 Java methods in our study. We sourced these by randomly sampling 60 methods from the test set of the `funcom-java-long` dataset in our experiment in the previous sections, then removing samples where CCE to `use-seq` led to the model generating the identical summary. The sample size of 50-60 is suitable because it is large enough to provide a meaningfully representative sample, but small enough to be evaluated within 90 minutes, after which fatigue bias may become a major factor[71].

Participants We used the Prolific[†] platform to recruit 29 English-speaking participants who were at least 25 years of age, hold a degree in Computer Science or Engineering, and self-report experience with Java.

7 | HUMAN STUDY RESULTS

This section discusses the results of our human study. Figure 2 shows the number of responses from all users for each quality attribute. The way to read this figure is that out of 1624 total survey responses for each question (29 participants x 56 methods), e.g., 850 preferred `use-seq` overall versus 650 for CCE (52% vs 40%, with the balance undecided). The human raters had a stronger preference for `use-seq` than CCE in overall subjective judgment.

One reason appears to be higher levels of accuracy for `use-seq`. The strongest difference is evident for Q1 about accuracy, where over 55% of responses favored `use-seq` and 34% favored CCE, with only around 11% unable to decide. The accuracy responses seem associated with completeness, where `use-seq` also performs well. This result aligns with our experimental

[†] https://www.prolific.co/

results in Section 5.4, where `use-seq` tends to find more words to explain what the components of the subroutine do, such as using the word "control" to modify "fps" in Example 1, Table 9.

The `use-seq` approach does not always perform well in completeness, as Example 2, Table 9 shows. But in general, the errors `use-seq` makes tend to be in generating summaries that are too verbose. CCE outperforms `use-seq` by a 2-1 margin in conciseness. In addition, similarity to the reference does not seem to be associated with higher overall preference. The users did tend to view `use-seq` as generating summaries more similar to the reference than CCE, though the difference is the lowest of any quality attribute, and the users did not see the reference until the last question page.

To provide another view of the data, we also grouped the responses by participant (Figure 3). We define "group by participant" as the number of times that participant responded with each option. The mean value is visible as the red line in the boxplots, and the median value is the black middle line. Raw minimum, maximum, median, and mean values for `use-seq` are in the table. All participants rated all 56 methods. An "average" participant rated 32 (median) or 30.58 (mean) of the methods as preferring `use-seq` in terms of accuracy, compared to fewer than 20 for CCE, and undecided for six.

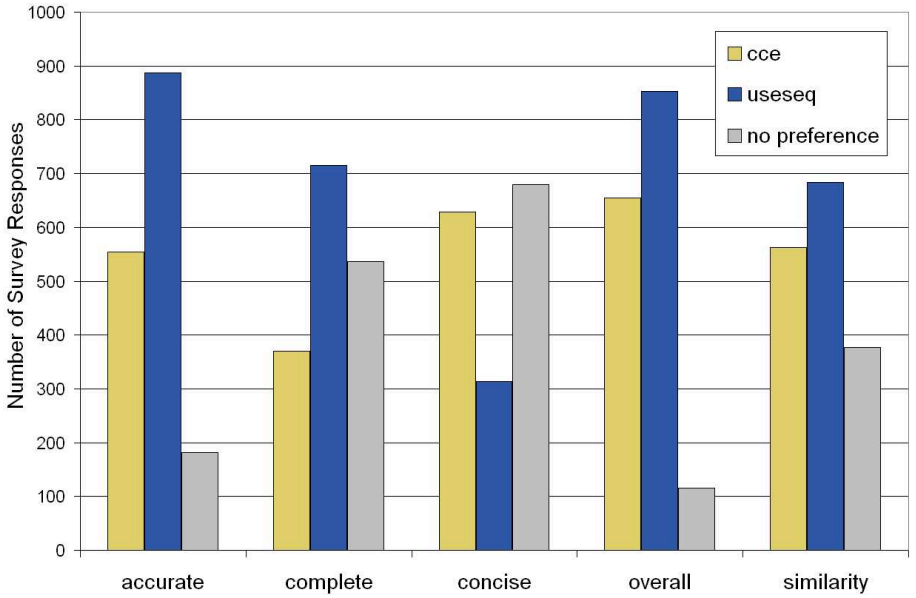


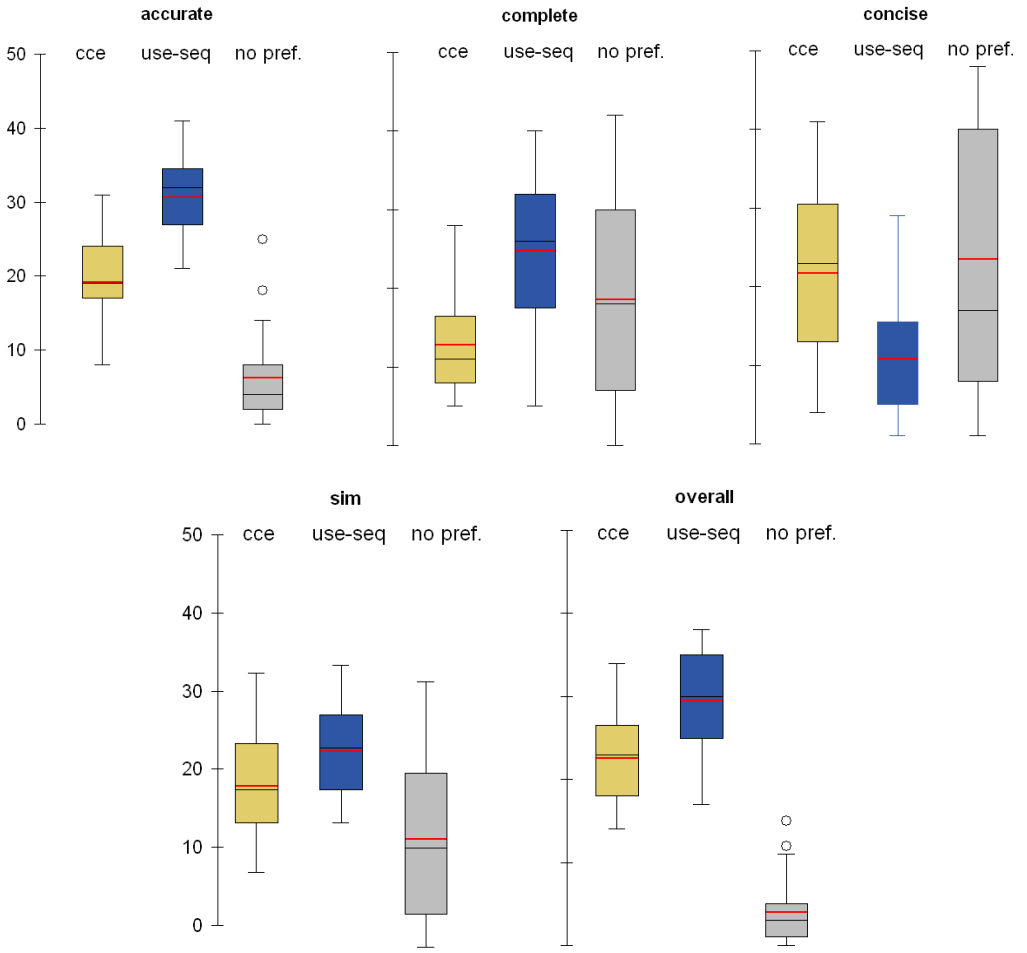
FIGURE 2 Aggregate survey responses. Each column indicates the total number of survey replies for a quality attribute and summaries from one loss function. With 29 participants and 56 functions, there are 1624 responses per attribute. For example, there were ∼900 responses in which participants preferred `use-seq` in terms of accuracy versus ∼550 for CCE.

The results broadly concur with the aggregate survey responses. The overall preference responses lean towards `use-seq`, where a typical user preferred `use-seq` in 30 (median) or 29.48 (mean) of 56 methods. The accuracy, completeness, and overall similarity likewise lean towards `use-seq`, though again conciseness does not, and again similarity only slightly favors `use-seq`.

As an additional check, we performed a Friedman paired statistical test for each of the quality attributes, grouped by participant. We report $Q_{observed}$, $Q_{critical}$, and p values for these tests in Figure 3. The Friedman test is the correct test because there are three sets of values, and each set of values are paired because they are tied to a specific participant over the same set of methods. It is also a non-parametric test, making it suitable for use in preference rankings where preference itself is measurable, but the degree of difference in preference is difficult to measure[72]. In general, strongest statistical significance occurred for the overall and accuracy questions, which supports the general conclusion that participants preferred summaries from `use-seq` in part due to the higher overall accuracy.

7.1 | Threats to Validity

We also divide the key threats to validity in this study into three different categories as in the previous section. The internal threat to validity involves our choice for the model, the subroutines, and the participants. We attempted to mitigate these risks by



	min	max	med	mean	Qobs	Qcrit	p
accurate	21	41	32	30.58	41.45	5.99	<0.01
complete	5	40	26	24.69	15.52	5.99	<0.01
concise	1	29	8	10.83	13.50	5.99	<0.01
overall	17	38	30	29.48	46.41	5.99	<0.01
similarity	9	34	24	23.59	7.122	5.99	<0.01

FIGURE 3 Table shows a statistical summary of responses for `use-seq` grouped by participant, followed by Friedman significance test results. For example, an "average" participant, in terms of accuracy, preferred 30.58 summaries by `use-seq` compared to ∼19 for CCE, and had no preference for ∼6. Boxplots show comparisons grouped by participant for all questions.

choosing a random sample of subroutines and human programmers large enough to provide a representative sample, but the risk remains that different results are possible with different inputs. Another internal threat to validity is that the participants might fake the information in the online platforms such as Prolific or the participants might just click through without reading the information carefully. We mitigate this threat by manually inspecting the time that the participants spent on each question. We do not include the programming knowledge test because those questions can be easily answered by using AI-based tools such as ChatGPT[73]. The construct threats to validity include the survey interface design and the wording of the questions. We attempted to mitigate threats from the survey interface and questions by adhering to practice in related work, it is possible that differently-worded questions or interface design could change the study results. The external threat to validity is that the programmer that we hired to evaluate the summary might not be good at Java programming. We mitigated this threat by setting the constraint that the participants should be at least 25 years old, which eliminates the undergraduate students.

8 | DISCUSSION

The integration of semantics similarity to the loss function is not limited to USE score. The use of the USE scores demonstrated that it is possible to integrate the semantic similarity into the loss function to give the reward for the correct prediction and penalize the wrong prediction. The future experiments include exploring the use of different semantic metrics between reference summary and predicted summary such as CrystalBLEU[44] and the use of semantic metrics between source code and predicted summary such as SIDE[45].

Our approach is likely to have an impact beyond code summarization. Within the field of Software Engineering, several tasks involve learning representations of code and writing or modifying text, such as bug report description and triage, test generation, and automatic privacy notification. In fields beyond SE, `use-seq` may have applications in numerous areas of text generation, such as neural machine translation and general tasks involving fine-tuning of LLMs. We demonstrated how our approach can be used in fine-tuning LLMs for one task, with other areas as future work.

9 | CONCLUSION

This paper moves the state-of-the-art forward in five ways:

1. We introduce a procedure for using a semantic similarity loss function (that we call `use-seq`), that we designed for the software engineering task of code summarization.
2. We evaluate our approach with four purpose-built code summarization models over datasets in two programming languages against three baselines (CCE, BLEU, SimiLE). We show how `use-seq` achieves improvements in several conditions.
3. We propose and implement code summarization as fine-tuning of two industrial language models.
4. We evaluate our approach over a Java dataset against the state-of-the-practice CCE using automated metrics.
5. We perform a study with 29 human programmers to evaluate summaries for 54 Java methods. We compare and contrast summaries generated by LLaMA with `use-seq` versus LLaMA with CCE.

Overall, we found that `use-seq` improves purpose-built code summarization approaches by 2-3% when measured by automated metrics METEOR and USE, and up to 12% when measured by BLEU. This is a strongly positive result considering that the improvement 1) is consistent over multiple approaches and datasets, and 2) comes at practically no cost or additional training procedure complexity. The `use-seq` loss function may be used as a drop-in replacement for CCE, unlike baselines which require additional steps.

We have designed an evaluated our approach for the problem of code summarization in the domain of software engineering. Key considerations we made for our target problem include 1) computing USE to compare summaries, which is the semantic similarity measure that recent literature[10] found is most associated with human programmer judgments of summary similarity, 2) we mask semantic similarities to avoid inappropriate penalties for correct word predictions when the sequence similarity is overall poor (and avoid rewarding incorrect words when the sequence similarity is good), and 3) we do *not* include a length penalty as baselines from NLP do, since people tend to prefer accuracy to conciseness.

To encourage maximum reproducibility and accessibility of our research, we provide our implementation code, training data, evaluation scripts, and further results via an online repository in Code Availability and Data Availability Section. Note that the full code of the examples provided in Section 5.4 of this paper are available by using the ID number under the example method name with the *fid* index in the `funcom-java-long` dataset we provide in Data Availability Section.

ACKNOWLEDGMENTS

This work is supported in part by NSF CCF-2100035 and CCF-2211428. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors

CONFLICT OF INTEREST

The authors declare no potential conflict of interests.

DATA AVAILABILITY

The funcom-python dataset that we propose is in our APCL Huggingface repository:
`https://huggingface.co/datasets/apcl/funcom-python/tree/main`
Also, we released the prediction files and models on LLM in our APCL Hugginface repository:
`https://huggingface.co/apcl/funcom_useloss/tree/main`

CODE AVAILABILITY

We release our code for experiments in our APCL Github repository, `https://github.com/apcl-research/funcom-useloss`

REFERENCES

1. Haiduc S, Aponte J, Moreno L, Marcus A. On the use of automated text summarization techniques for summarizing source code. In: IEEE. 2010:35–44
2. Robillard MP, Marcus A, Treude C, et al. On-demand developer documentation. In: IEEE. 2017:479–483
3. LeClair A, Jiang S, McMillan C. A neural model for generating natural language summaries of program subroutines. In: IEEE Press. 2019:795–806
4. LeClair A, Haque S, Wu L, McMillan C. Improved code summarization via a graph neural network. In: 2020:184–195
5. Haque S, LeClair A, Wu L, McMillan C. Improved Automatic Summarization of Subroutines via Attention to File Context. *International Conference on Mining Software Repositories.* 2020. doi: 10.1145/3379597.3387449
6. Tang Z, Shen X, Li C, et al. AST-trans: code summarization with efficient tree-structured attention. In: 2022:150–162
7. MacNeil S, Tran A, Hellas A, et al. Experiences from using code explanations generated by large language models in a web software development e-book. In: 2023:931–937
8. Ross SI, Martinez F, Houde S, Muller M, Weisz JD. The programmer's assistant: Conversational interaction with a large language model for software development. In: 2023:491–514
9. Wieting J, Berg-Kirkpatrick T, Gimpel K, Neubig G. Beyond BLEU: Training Neural Machine Translation with Semantic Similarity. In: 2019:4344–4355
10. Haque S, Eberhart Z, Bansal A, McMillan C. Semantic similarity metrics for evaluating source code summarization. In: 2022:36–47
11. Touvron H, Lavril T, Izacard G, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971.* 2023.
12. Taori R, Gulrajani I, Zhang T, et al. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tloen/alpaca-lora; 2023.
13. Wang E. Alpaca-LoRA. https://github.com/tatsu-lab/stanford_alpaca; 2023.
14. Hu EJ, Wallis P, Allen-Zhu Z, et al. LoRA: Low-Rank Adaptation of Large Language Models. In: 2023.
15. Alon U, Brody S, Levy O, Yahav E. code2seq: Generating sequences from structured representations of code. *International Conference on Learning Representations.* 2019.
16. Alon U, Zilberstein M, Levy O, Yahav E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages.* 2019;3(POPL):1–29. doi: 10.1145/3290353
17. Nie P, Rai R, Li JJ, Khurshid S, Mooney RJ, Gligoric M. A framework for writing trigger-action todo comments in executable format. In: ACM. 2019:385–396
18. Haldar R, Wu L, Xiong J, Hockenmaier J. A Multi-Perspective Architecture for Semantic Code Search. *arXiv preprint arXiv:2005.06980.* 2020.
19. Ahmad WU, Chakraborty S, Ray B, Chang KW. A Transformer-based Approach for Source Code Summarization. *arXiv preprint arXiv:2005.00653.* 2020.
20. Feng Z, Guo D, Tang D, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In: 2020:1536–1547
21. Bansal A, Haque S, McMillan C. Project-level encoding for neural source code summarization of subroutines. In: IEEE. 2021:253–264
22. Zügner D, Kirschstein T, Catasta M, Leskovec J, Günnemann S. Language-Agnostic Representation Learning of Source Code from Structure and Context. In: 2021.
23. Liu S, Chen Y, Xie X, Siow JK, Liu Y. Retrieval-Augmented Generation for Code Summarization via Hybrid {GNN}. In: 2021.
24. Mastropaolo A, Scalabrino S, Cooper N, et al. Studying the usage of text-to-text transfer transformer to support code-related tasks. In: IEEE. 2021:336–347.
25. Kuang L, Zhou C, Yang X. Code comment generation based on graph neural network enhanced transformer model for code understanding in open-source software ecosystems. *Automated Software Engineering.* 2022;29(2):43. doi: 10.1007/s10515-022-00341-1
26. Li Z, Wu Y, Peng B, et al. Setransformer: A transformer-based code semantic parser for code comment generation. *IEEE Transactions on Reliability.* 2022. doi: 10.1109/TR.2022.3154773
27. Khan JY, Uddin G. Automatic code documentation generation using gpt-3. In: 2022:1–6.
28. Ahmed T, Devanbu P. Few-shot training LLMs for project-specific code-summarization. In: 2022:1–5.
29. Gu J, Salza P, Gall HC. Assemble foundation models for automatic code summarization. In: IEEE. 2022:935–946.
30. Su CY, Bansal A, Jain V, Ghanavati S, McMillan C. A Language Model of Java Methods with Train/Test Deduplication. In: ESEC/FSE 2023. Association for Computing Machinery 2023; New York, NY, USA:2152–2156
31. Gao S, Gao C, He Y, et al. Code Structure–Guided Transformer for Source Code Summarization. *ACM Transactions on Software Engineering and Methodology.* 2023;32(1):1–32.
32. Geng M, Wang S, Dong D, et al. Interpretation-based Code Summarization. In: 2023.
33. Zhang M, Zhou G, Yu W, Huang N, Liu W. Ga-scs: Graph-augmented source code summarization. *ACM Transactions on Asian and Low-Resource Language Information Processing.* 2023;22(2):1–19.
34. Gao Y, Zhang H, Lyu C. EnCoSum: enhanced semantic features for multi-scale multi-modal source code summarization. *Empirical Software Engineering.* 2023;28(5):126.
35. Wang Z, Yu X, Feng Y, Zhao D. An Intra-Class Relation Guided Approach for Code Comment Generation. In: 2023:1291–1303.
36. Geng M, Wang S, Dong D, et al. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. 2024.
37. Song X, Sun H, Wang X, Yan J. A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques. *IEEE Access.* 2019.

38. Jin X, Larson J, Yang W, Lin Z. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601.* 2023.
39. Sun W, Fang C, You Y, et al. Automatic Code Summarization via ChatGPT: How Far Are We?. *arXiv preprint arXiv:2305.12865.* 2023.
40. Wang Y, Wang W, Joty S, Hoi SC. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In: Moens MF, Huang X, Specia L, Yih SWt., eds. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*Association for Computational Linguistics 2021; Online and Punta Cana, Dominican Republic:8696–8708
41. Bender EM, Gebru T, McMillan-Major A, Shmitchell S. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In: FAccT '21. Association for Computing Machinery 2021; New York, NY, USA:610–623
42. Papineni K, Roukos S, Ward T, Zhu WJ. BLEU: a method for automatic evaluation of machine translation. In: Association for Computational Linguistics. 2002:311–318
43. Banerjee S, Lavie A. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In: 2005:65–72.
44. Eghbali A, Pradel M. CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code. In: ASE '22. Association for Computing Machinery 2023; New York, NY, USA
45. Mastropaolo A, Ciniselli M, Di Penta M, Bavota G. Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization. In: ICSE '24. Association for Computing Machinery 2024; New York, NY, USA
46. Ranzato M, Chopra S, Auli M, Zaremba W. Sequence level training with recurrent neural networks. In: 2016.
47. Pasunuru R, Bansal M. Multi-Reward Reinforced Summarization with Saliency and Entailment. In: 2018:646–653
48. Chen Y, Lu X. Deep category-level and regularized hashing with global semantic similarity learning. *IEEE transactions on cybernetics.* 2020;51(12):6240–6252. doi: 10.1109/TCYB.2020.2964993
49. Nakatani Y, Kajiwara T, Ninomiya T. Comparing BERT-based Reward Functions for Deep Reinforcement Learning in Machine Translation. In: 2022:37–43.
50. Yasui G, Tsuruoka Y, Nagata M. Using semantic similarity as reward for reinforcement learning in sentence generation. In: 2019:400–406
51. Cer D, Yang Y, Kong Sy, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175.* 2018.
52. Korbak T, Shi K, Chen A, et al. Pretraining language models with human preferences. *arXiv preprint arXiv:2302.08582.* 2023.
53. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in software engineering.* Springer Science & Business Media, 2012.
54. Hellendoorn VJ, Sawant AA. The growing cost of deep learning for source code. *Communications of the ACM.* 2021;65(1):31–33. doi: 10.1145/3501261
55. Xu FF, Alon U, Neubig G, Hellendoorn VJ. A systematic evaluation of large language models of code. In: 2022:1–10
56. Wieting J, Gimpel K. ParaNMT-50M: Pushing the Limits of Paraphrastic Sentence Embeddings with Millions of Machine Translations. In: 2018:451–462
57. Wu Y, Schuster M, Chen Z, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144.* 2016.
58. LeClair A, McMillan C. Recommendations for Datasets for Source Code Summarization. In: 2019:3931–3937
59. Allamanis M. The adverse effects of code duplication in machine learning models of code. In: 2019:143–153
60. Shi L, Mu F, Chen X, et al. Are We Building on the Rock? On the Importance of Data Preprocessing for Code Summarization. In: ESEC/FSE 2022. Association for Computing Machinery 2022:107–119
61. Haque S, Bansal A, Wu L, McMillan C. Action Word Prediction for Neural Source Code Summarization. *28th IEEE International Conference on Software Analysis, Evolution and Reengineering.* 2021. doi: 10.1109/SANER50967.2021.00038
62. Roy D, Fakhoury S, Arnaoudova V. Reassessing Automatic Evaluation Metrics for Code Summarization Tasks. In: 2021
63. Radford A, Wu J, Child R, et al. Language models are unsupervised multitask learners. *OpenAI blog.* 2019;1(8):9.
64. Karpathy A. nanoGPT: The simplest, fastest repository for training/finetuning medium-sized GPTs.. https://github.com/karpathy/nanoGPT; 2023.
65. Su CY, McMillan C. Distilled GPT for source code summarization. *Automated Software Engineering.* 2024;31(1):22.
66. Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K. Towards automatically generating summary comments for java methods. In: ACM. 2010:43–52
67. Ferretti C, Saletta M. Naturalness in Source Code Summarization. How Significant is it?. In: IEEE. 2023:125–134.
68. Rani P, Blasi A, Stulova N, Panichella S, Gorla A, Nierstrasz O. A decade of code comment quality assessment: A systematic literature review. *Journal of Systems and Software.* 2023;195:111515.
69. Dell N, Vaidyanathan V, Medhi I, Cutrell E, Thies W. " Yours is better!" participant response bias in HCI. In: 2012:1321–1330
70. Chyung SY, Barkin JR, Shamsy JA. Evidence-based survey design: The use of negatively worded items in surveys. *Performance Improvement.* 2018;57(3):16–25. doi: 10.1002/pfi.21749
71. Sievertsen HH, Gino F, Piovesan M. Cognitive fatigue influences students' performance on standardized tests. *Proceedings of the National Academy of Sciences.* 2016;113(10):2621–2624. doi: 10.1073/pnas.1516947113
72. Sheldon MR, Fillyaw MJ, Thompson WD. The use and interpretation of the Friedman test in the analysis of ordinal-scale data in repeated measures designs. *Physiotherapy Research International.* 1996;1(4):221–228. doi: 10.1002/pri.66
73. Ghorbani A, Cassee N, Robinson D, et al. Autonomy Is an Acquired Taste: Exploring Developer Preferences for GitHub Bots. In: ICSE '23. IEEE Press 2023:1405–1417